Bringing HLA to the Web: The Incorporation of Enterprise Java Technology into HLA Federations

Conrad Housand
chousand@aegistg.com

Bill Hudgins
bhudgins@aegistg.com

The AEgis Technologies Group, Inc.
12565 Research Parkway
Suite 390
Orlando, Florida 32826
407-380-5001

Keywords:
Java, Servlets, EJB, Federation Management.

ABSTRACT: *As the Java platform evolves and matures, many opportunities arise for the integration of Web technology into HLA federations. In particular, Java Servlets and Enterprise Java Beans (EJBs) provide attractive mechanisms for constructing Web Browser-based interfaces to simulation data.*

This paper describes techniques for using Java Servlets, EJBs, and Java Federate application components to extend web server functionality in support of browser-based federation management. In the approach taken here, Servlets are used as an RTI interface for a web server, thereby making the web server a working HLA federate. To provide maximum flexibility in managing federation activities, small Java-based agent federates run on all machines on the simulation network, and communicate with the web server over the RTI via a management-specific "meta-federation." The Java "meta-federates" can then perform activities such as launching simulation federates, collecting low-level performance data, etc. These activities are controlled and monitored with a web browser acting as a client to the RTI-aware web server. This framework would, for example, allow management of multiple federations in diverse locations via the Internet from a palmtop-sized computer.

While the specific application of the technology presented here emphasizes federation management, the underlying techniques are generally extensible to the creation of a web-based interface to any kind of simulation data.

1. Introduction

Recent additions to the Java platform include a number of APIs collectively referred to as "Enterprise Java." These technologies build upon the core Java specification to provide a range services commonly used in distributed computing, and of particular benefit to web-browser based applications. Taken together, the Enterprise Java APIs provide a standard, consistent, and platform independent framework for building scalable multitier applications (see [1] for a thorough description of Java Enterprise technologies). Moreover, Enterprise Java presents an attractive mechanism for extending the potential for interaction with HLA Federations to any platform or location in which a web browser can be used. In effect, the integration of Enterprise Java and HLA make possible a notional "online federation" in which browser-based applications could view and interact with HLA simulation data over an intranet or internet.

2. Background

While the Enterprise Java APIs provide a whole range of services which are of use in distributed computing, two technologies in particular are central to the techniques which will be discussed shortly: Java Servlets and Enterprise Java Beans (EJBs). We start by providing some background on these technologies.

2.1 Servlets

Java Servlets are similar in nature to the more familiar Java Applets. Applets are fundamentally Java programs which support a standard API and run within the context of a web browser (i.e., they extend the functionality of a web browser). Similarly, Servlets can be thought of as Java applications which run in the context of a web server, and extend the capabilities of that server. A Servlet depends upon a set of services provided by the web server; these services denote a *Web Container*, which acts as an intermediary between the browser HTTP requests and any Servlets running on the server. A particular Servlet is initialized and invoked by a web server when a client (web browser) requests a particular URL. Typically, the Servlet is then responsible for the generation of dynamic content, which is then returned to the client. This is accomplished via standard services which are provided by the Servlet, and which are utilized by any web server which supports the Servlet API. The manner in which Servlets are deployed/installed on a web server is somewhat specific to the particular web server software being used, but the implementation of the Servlet itself is not specific to a particular web server. Figure 2.1 depicts the relationship between the client application, web container, and servlet object.

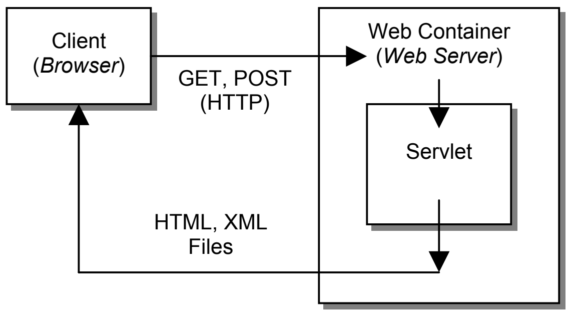

Figure 2.1 Relation of client and Servlet objects.

Fundamentally, Servlets can be implemented by constructing a Java class which inherits from the *javax.servlet.Servlet* class with appropriate method overrides to insert any code necessary to do HTML content generation. In practice, however, the *javax.servlet.http.HttpServlet* class provides a better foundation for any Servlet intended to generate HTML content using the HTTP protocol. A brief summary of some of the more important methods of this class follows:

init() : Called by the web container when the servlet is loaded for the first time. An override of this method can be used to set up I/O resources or perform any other one-time housekeeping tasks.

destroy() : Called by the web container just before the Servlet object is destroyed (typically when the web server is shut down). Override this method to perform deinitialization tasks.

doGet(…) : Called by the web container in response to a client (web browser) request to retrieve (HTTP GET command) a web page. Dynamic HTML content generation, for example could be done in an override of this method. Arguments passed in to this method are used to return the dynamically generated content.

doPost(…) : Called by the web container in response to a client HTTP POST command. Override this method to receive input from HTML forms. Arguments passed in to this method are used to determine the values of fields in the submitted HTML form, and to return and content in response to the POST command.

Of particular note is the fact that Servlets persist between invocations; that is, once a Servlet is instantiated and running within a web server, it will remain so, even after termination of a client session. This means that a Servlet provides a suitable point at which an RTI interface may be inserted into a web server. This fact becomes the basis for techniques which will be discussed in the following Sections.

2.2 Enterprise Java Beans

Enterprise Java Beans are essentially foundations for building component-based distributed objects. Generally, the EJB foundation implements the low-level tasks which are common to all distributed objects (networking, transaction handling, security, etc.), while avoiding the details of application-specific logic. EJBs have a number of similarities with Servlets, in that they support a standard API which allows them to be platform and web-server independent, and generally exist within the context of the web server. The fundamental difference is that EJBs are distributed objects, and can be directly operated upon from within the client application as though they existed in the same process space. Servlets can only communicate with clients via some sort of intermediate content (e.g., HTML or XML files).

EJBs are, by design, reliant upon services provided by an *EJB Container*. The EJB Container acts as the intermediary between the EJB components, and client applications which are invoking methods of those components; its role is similar to that of the ORB in CORBA. In addition, the EJB Container provides a range of security, transaction, and object persistence services. Figure 2.2 depicts the relationship between the client application, EJB container, and EJB objects. The EJB Container is provided by the web server software. Examples of web servers which provide support for EJBs include IBM's WebSphere and BEA's WebLogic. It is interesting to note that an appropriate web server and set of EJBs could conceivably serve as the foundation for an HLA RTI.

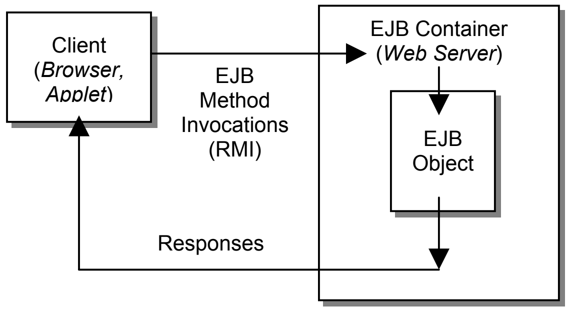
Client (*Browser, Applet*)
EJB Method Invocations (RMI)
EJB Container (*Web Server*)
EJB Object
Responses

Figure 2.2 Relation of client, EJB, and EJB container objects.

Implementation of EJBs is done by creating three java constructs for each bean. The first is an interface, called the *home interface*, which is accessed by remote clients; it's used to create or find EJBs objects of this class. This interface inherits from the *javax.ejb.EJBHome* interface. The second required interface, called the *remote interface*, is used specify the methods of the EJB that will be exported to remote clients. This interface inherits from *javax.ejb.EJBObject* interface. The last required class is the implementation of the bean itself. Definition of it's methods must adhere to certain strict conventions; see reference [1], [2], and [3] for complete details regarding EJB implementation.

After deploying the EJB, a remote client (an applet running in a web browser, for instance) can create an instance of the EJB, or look up a reference to an existing EJB instance already running in the server. In either case the client application will receive an instance of an object which implements the remote interface for that particular EJB, and can be used to remotely invoke the methods of the corresponding EJB instance running on the server.

EJBs fall into two broad categories, depending on the particular services they provide, whether they are persistent in nature, and how they interact with clients.

2.2.1 Session Beans

Session Beans generally encapsulate services which are used directly by client applications, and usually only exist for the duration of the client-server session. Beans of this type can be further categorized by the manner in which they handle state information across method invocations. A Session Bean which does not change state across method calls is said to be a *stateless* session bean. Stateless session beans can be used concurrently by multiple remote clients without possibility of data corruption. *Stateful* session beans, by contrast, maintain state information which is subject to change by remote method invocations. Stateful session beans are not typically used by more than one client at a time.

In general, session beans provide an easy means of establishing a connection between a client process (applet) and one or more RTI-enabled servlets running within the server.

2.2.2 Entity Beans

Entity Beans encapsulate data which persists across client sessions. Typically, this refers to data stored in a database. The services which serialize entity beans to and from persistent storage are supplied by the EJB container, and in fact can be made transparent to both the client and entity bean. Like session beans, entity beans are ultimately distributed objects; their methods may be remotely called by client applications in order to effect state changes. For certain types of entity beans, the EJB container automatically updates persistent storage to reflect changes in the bean's state data, and creates and remove instances of the bean as necessary. This arrangement is referred to a *container-managed persistence*. By contrast, the bean itself may internally save and restore its state information to and from persistent storage upon notification from the container. This is referred to as *bean managed persistence*.

Since entity beans can provide a remote interface to persistent information stored in a server-side database (in particular, a *relational* database), they are a good candidate for constructing a general means of extending the visibility of FOM data to client (browser) applications. This idea will be examined in more detail in subsequent sections.

3. The Web Server Federate

For purposes of Federation management via remote web-based applications, making the web server a working HLA federate seems an appropriate strategy. This would in principle allow the web server to perform federation management activities on behalf of a web-based client application, to communicate over HLA with management-specific remote (agent) federates, and to act as a data collector for management-specific FOM data (or simulation data in general). Due to the certain restrictions imposed on EJBs (they must be strictly single-threaded, for instance), it makes the most sense to HLA-enable a web server by placing the RTI interface code in a Servlet.

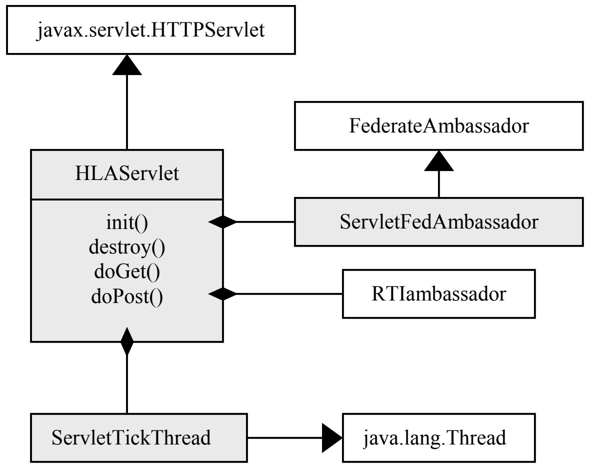

Figure 3.1 Design of HLA-enabled Servlet

Figure 3.1 illustrates the fundamental requirements for RTI enabling a servlet class (the shaded boxes indicate classes which must be created, white boxes denote components supplied by either the java platform or the RTI implementation). Firstly, the servlet class must provide any overrides of the base class which are necessary to respond to HTTP commands. These are indicated in Figure 3.1 by the *doGet()* and *doPost()* methods in the *HLAServlet* class. Next, the servlet class must contain references to an instance of the *RTIambassador* class and a class which inherits the *FederateAmbassador* class. The *ServletFed-Ambassador* in Figure 3.1 denotes the derivation of the *FederateAmbassador* class, with any necessary overrides as required to receive the desired RTI callbacks. Lastly, the servlet class must contain some mechanism for ticking the RTI, even in the absence of client requests which invoked the *doGet()* or *doPost()* methods. This is accomplished by creating a separate thread (the *ServletTickThread* class), which continually calls the *tick()* method of the *RTIambassador* instance referenced by the *HLAServlet* class.

Federate initialization is performed in the override of the servlet *init()* method. For example, this is the location where the servlet attempts to create and join a federation execution. It's also the place where any name-to-handle lookups are performed, where any declarations are performed, and where any servlet state data is initialized. Similarly, the override of the destroy() method is where the servlet resigns from a federation.

As a concrete example, consider a simple servlet which subscribes to attributes of the MOM *Manager.Federate* class and collects attribute reflections for instances of this class. This allows the servlet to dynamically generate an HTML page with a list of the joined federates and values for corresponding attributes of the *Manager.Federate* instance whenever a web browser requests the associated URL. Implementation of this simple federate-servlet generally involves the steps described above with the following specific additions: Attributes of the *Manager.Federate* class should be subscribed to in the override of the *init()* method. The *ServletFedAmbassador* implementation should provide code in the *discoverObjectInstance()*, *reflectAttributeValues()*, and *removeObjectInstance()* to maintain a vector of objects which encapsulate the information in attributes of the *Manager.Federate* class. The vector and the corresponding values can then be used in the *doGet()* method to generate a list of joined federates and values for attributes corresponding *Manager.Federate* object instances. Lastly, code can be added to the *ServletTickThread* class to periodically request an attribute update of all instances of the *Manager.Federate* class.

Alternatively, this servlet could store the reflected attributes of *Manager.Federate* instances in corresponding Entity EJBs, and suppress any HTML content generation. In this case, the client applications would directly access remote interfaces to the Entity EJBs and perform the data presentation logic in the client application itself. This arrangement would make sense when the clients were Java applets, which could instantiate and manipulate remote interfaces to the EJBs maintained by the servlet. The distinction between these two methods will be discussed in Section 5.

4. Management-specific Federations

While the Management Object Model (MOM) provides a means for interacting with a running federation to perform basic management activities, it does not provide a mechanism for remotely initializing federates from some central location before they have joined a federation, or for collecting low-level network traffic and configuration details.

These requirements could, in principle, be addressed by placing EJB client applications on all nodes which might host a simulation federate, and then using EJB enabled communications between the servers and clients. Clients on each node could potentially collect the required data or spawn a simulation federate process on that node.

In fact, though, this arrangement has certain drawbacks. Firstly, the client-server association between the EJB container and client applications running on other nodes is not well suited for server-side initiation of management activities. Second, this requires that targets of management messages be clients employing EJB technology. This could potentially preclude many scenarios in which a simulation application itself would become a consumer of management messages.

A more suitable arrangement can be accomplished by adopting the notion of a "meta-federation": i.e., a federation which facilitates communication between the web server and lightweight agent federates (or potentially modified simulation applications themselves) which are responsible for collecting the desired network data or spawning and initializing the "real" simulation federates. In this case, the agent applications assume more of a peer-to-peer relationship with the web server than in the case of EJB-based clients. Generally, the lifespan of this federation would be long compared to the simulations federations which are spawned by the agent federates. The FOM of the meta-federation would be defined exclusively for management data; this could include classes/interactions for passing network traffic or CPU load data, issuing commands to start a remote simulation federate, or performing remote communication with a spawned simulation federate from some central location via a pipe between the agent process and the spawned process. Conceivably, the FOM might even contain classes for passing java bytecode and serializing java objects across the RTI to be loaded and run within the virtual machine of one of the agent federates.

Figure 4.1 depicts the associations between the web server federate, and a typical agent federate and spawned simulation federate.

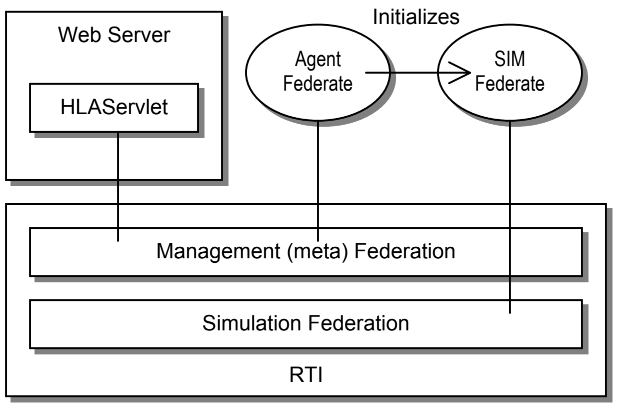

Figure 4.1 Associations between web server, agent, and simulation federates.

5. Implementation Approaches

We now describe two ways in which the ideas and technologies described so far can be integrated to achieve a centralized, web-based federation management system. The primary distinction between the two methods lies in the location of the presentation logic which generates views of federation data and processes user input for initiation of management activities. However, a number of concepts which are common to both methods warrant discussion first.

We've already mentioned that Entity EJBs are really distributed-object encapsulations of persistent data, which is usually stored in a database. In fact, the relational database model lends it self well to the storage, retrieval and analysis of HLA object model data. As a result, Entity EJBs provide a natural and well-suited mechanism for the storage and retrieval of HLA data to and from persistent databases. Moreover, because the Entity EJBs are distributed in nature, they provide an automatic means of making that data available to remote, web-based applications. To continue with the example of the HLA Servlet which maintains a list of *Manager.Federate* object instances, consider a database on the server which contains a table which has columns corresponding to the attributes of the *Manager.Federate* class, and an associated Entity EJB which encapsulates the *Manager.Federate* class. If the servlet creates one instance of this bean for every discovered instance of *Manager.Federate*, then the associated database table will have one row for every *Manager.Federate* instance, and the columns of each row will have values which mirror the values of the attributes for the corresponding instance of the *Manager.Federate* class. More importantly, applets running in remote web browsers will be able to query the members of these EJBs, and thus be able to view runtime FOM data.

This idea could, of course, be extended to FOM data in general. In this sense, agent federates could assume the role of distributed data collectors by joining the simulation federation and capturing the relevant FOM data as entity EJBs. Browser-based client applications could then be extended to include capabilities for simulation data analysis or AAR.

5.1 Server-centric

As its name suggests, the Server-centric scenario places presentation logic on the server. In general, the client communicates with the server in this scenario mostly through HTTP (i.e., by receiving HTML pages and posting info from forms on those pages). Generation of the HTML content is performed by servlets, which either interact directly with the RTI via an RTIambassador, or access federation data via Entity Java Beans which encapsulate FOM classes and are updated by other RTI-aware servlets. User input is captured via HTML forms; the input data is received by the appropriate Servlet in the doPost() method. This scenario has the advantage of placing minimal computational requirements on the client. However, the user interface is constrained by the limits of HTML content.

Figure 5.1 illustrates the relation between the components of this scenario. One Servlet is responsible for performing dynamic HTML content generation. This servlet communicates with the RTI-enabled servlet via a Session EJB to request management actions. Similarly, it receives management data (meta-federation FOM) via multiple Entity EJBs (corresponding to FOM class instances or interactions) which are updated by the RTI-enabled servlet.

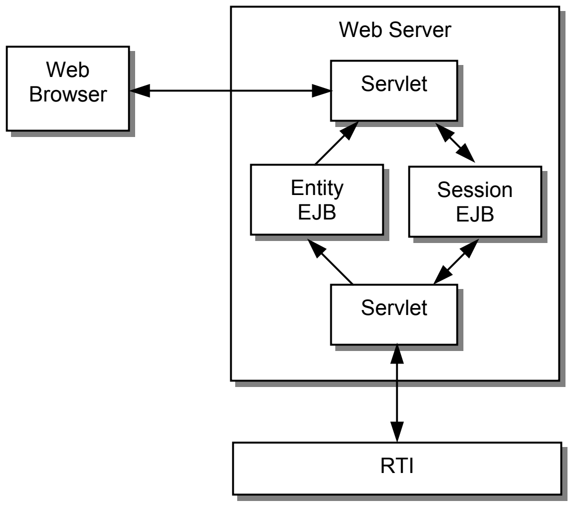

Figure 5.1 Server-centric architecture

5.2 Browser-centric

In the browser-centric scenario, a great deal of presentation logic is moved to the client. Typically this is done by via a java applet which runs in the web browser, and acts as a direct client of Entity and Session EJBs. In this sense, the web client effectively has direct access to the FOM data. The details of how this data is analyzed, reduced, and presented to the user are coded into the applet running in the web browser. User input is routed to appropriate methods in session beans. The primary advantage of this arrangement is that the user interface can be much more sophisticated; java applets can make use of any java GUI component available to a full java application.

Figure 5.2 illustrates the relation between the components of the browser-centric scenario. An applet running within the web browser is a client of both entity and session EJBs. Session EJBs provide the interface between the client-side applets and RTI-enabled servlet which allows for applet-initiated management activities and requests for management information. Entity EJBs encapsulate management data from the meta-federation FOM; their values are updated by the RTI-enabled servlet and monitored by the applet.

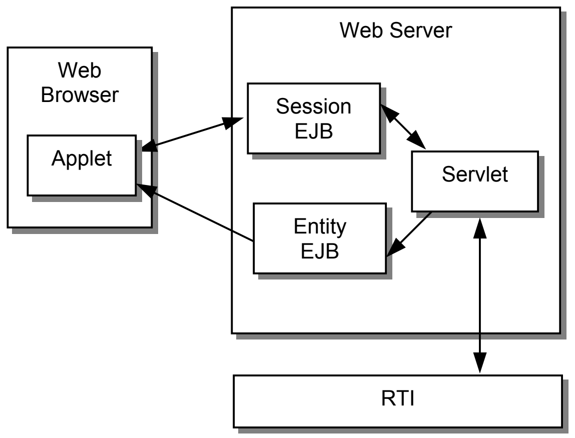

Figure 5.2 Browser-centric architecture

6. Summary

We have described here a foundation for extending HLA federation management capabilities to web browser based applications by employing Enterprise Java technology. While the example discussed here has focused on mechanisms intended to support federation management, the techniques are generally applicable to the construction of a web-based interface to any kind of HLA simulation data. This effectively allows any platform supporting a web browser and Internet connectivity to interact with HLA federations.

7. References

[1] “Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition.” Sun Microsystems, March 22, 2000.
[2] “The Java 2 Enterprise Edition Developer’s Guide.” Sun Microsystems, May, 2000.
[3] Flanagan, David, et al.: Java Enterprise in a Nutshell, O’Reilly and Associates, 1999.
[4] Pawlan, Monica: “Writing Enterprise Applications with Java 2 SDK, Enterprise Edition.” Sun Microsystems, June, 2000.

Author Biographies

CONRAD HOUSAND is a Software Engineer with The AEgis Technologies Group, Orlando, FL., where he is involved in the development of a number of HLA simulation tools.